Pythonic Research

Making Data Easier with Python

Willem Klumpenhouwer

University of Calgary

June 17, 2015

Outline

Section 1

Setting Up Shop

Where are We?

Credit Where Credit Is Due

Some useful content and structure taken from
http://www.aleax.it/goo_py4prog.pdf

Some excellent resources are available at Software Carpentry:
http://software-carpentry.org/lessons.html

What This Workshop **Is**

- For people with mild to advanced experience with another programming language
- An introduction to Python - no experience needed!
- Hoping to keep it clean and simple
- An overview of Python as a toolbox
- Designed to equip you with a tool belt of useful stuff you can use for almost any task
- **Inquiry based learning**

What This Workshop **Isn't**

- A rigorous in-depth programming course
- Taught by a veteran programmer
- Here to give you all the answers
- Here to make you a good programmer
- Here to teach you logic and problem solving

What Is Python?

- A very high-level, object oriented language
- Designed for ease of use, and clarity
- Emphasizes productivity through modularity, re-useability, and consistency
- Open source, cross platform, and community driven
- Well documented!
- Full of rich standard modules, 3rd-party modules

What Is Python? (more technical)

- Compiles to bytecode
- Everything is object-inherited
- Garbage collected
- Serialized, threads
- Simple and bare syntax, minimal ornamentation

Being *Pythonic*

Thinking Pythonic

To think Pythonic, think simple. Think direct. Think compact yet concise. **That** is how you solve your problems quickly, and with minimal fuss.

Coding Pythonic

Each statement should explain itself. It should complete a thought if possible. If you can accomplish this, you will have beautiful code.

Being Pythonic

```
import this
```

Python 2 and Python 3

There are two major versions of Python that are in use, with some subtle differences in them.

Python 2.7

- (Currently) more commonly used
- Questions are usually asked with this version in mind
- The "default" Python.

Python 3.x

- More consistency with built-in functions
- Different string formatting
- The future!

We will be using Python 3 for this workshop

Installing Python

Windows and Mac

Follow the link, and install. In this workshop, we are using Python 3.x
https://www.python.org/downloads/

Linux

Just use the terminal, and install with

```
sudo apt-get install python3
```

The Python Interpreter

Python runs with an interpreter (think: MATLAB), which accepts commands directly. If you see

>>>

Python is waiting for you to give it a command. If you see

...

You need to continue a statement (you are inside of a loop, for example). Try out some basic stuff:

```
>>> x=5
>>> x+2
7
>>> y="John"
>>> z=" Cleese"
>>> y+z
'John Cleese'
```

The Python Interpreter (cont'd)

If you have moved in scope, Python will require a double return to finish the loop...

```
>>> for x in range(1,5):
...         print(x)
...
1
2
3
4
```

Python Files

You can also run Python through files, like scripts. Just put your code into a file and run it

```
python3 foo.py
1
2
3
4
```

Python Editors

Since it's just a text file, you can use any text editor. IDLE (Windows) has a simple editor. My personal favourite is Gedit, a GNOME text editor that is also available on windows.

http://ftp.gnome.org/pub/GNOME/binaries/win32/gedit/

If it's not already installed on Linux:

```
sudo apt-get install gedit
```

Also try Notepad++

Scope, and a Technicality

Python interprets scope with indentation. Indents can be any size, but must be consistent. Let's get this out of the way now:

Use 4 spaces (not a tab) as indentation for every single Python program you write. Consistency, remember? Stack Overflow thanks you.

Set your editor *right now* to use 4 spaces, not tabs.

Section 2

The Fundamentals

Where are We?

Data Types

When declaring variables, you do not need to specify a type. Python interprets the type of the variable dynamically as you use it. You can also easily convert types

```
>>> x=4.0 #starts as a float
>>> str(x) #returns a string
'4.0'
>>> float('4.0') #returns a float
4.0
>>> int(float(str(x))) #float to string to float to int
4
```

You can check the type of a variable with:

```
>>> type(x)
<class 'float'>
```

Booleans

Boolean variables are essentialy 0 for False, and 1 for True. The boolean variables start with a capital letter (convention for most classes)

```
>>> type(False)
<class 'bool'>
>>> True+True
2
>>> False+True
1
```

Operators

```
>>> 3+2 #addition
5
>>> 3-2 #subtraction
1
>>> 3*2 #multiplication
6
>>> 3**2 #exponent
9
>>> 3/2 #float division
1.5
>>> 3//2 #integer division
1
>>> 3%2 #modulo
1
```

Built In Functions

```
>>> x=[2,3,4] #a list of numbers. More later...
>>> sum(x) #sum all the numbers
9
>>> abs(-2) #absolute value
2
>>> min(x) #find a minimum
2
>>> max(x) #find a maximum
4
>>> pow(4,2) #exponents, again
16
>>> round(2.6) #rounding, can specify precision
3
```

Basic Strings

```
>>> type("double quotes") #single or double?
<class 'str'>
>>> type('single quotes') #doesn't matter!
<class 'str'>
>>> "John" + " Cleese" #Concatenate - not fast!
'John Cleese'
>>> "John "*2 #Repetition
'John John '
```

Containers

```
>>> (4,'s') #tuple - immutable!
>>> [4,'s'] #list - mutable
>>> {4,"s"} #set - unique objects
>>> {"key1":4, "key2":'s'} #dict - key/value pairs
```

Lots of built-in functions for these. Can loop through containers, and also get length

```
>>> len([2,3,5,'s',4.2])
5
```

Sequences

Sequences are tuples, lists, or strings

```
>>> "spam"[0] #access index
's'
>>> "spam"[2:4] #take a slice
'am'
>>> "spam"[:3] #slice from beginning
'spa'
>>> "spam"[-1] #work backwards
'm'
```

Note: dictionaries and sets are not sequences.

Comparison

```
>>> 1<2 #strict inequality
True
>>> 1<=1 #loose inequality
True
>>> 1==1 #equal value
True
>>> 2 != 2.0 #not equal value
False
>>> 2 is 2.0 #identity operator
False
>>> 5 in [2,3,4] #inclusion
False
>>> 5 not in [2,3,4] #exclusion
True
```

Trust the Programmer

Use common sense when playing with containers and data types

- Can't sum a list with non-summable types
- Can't multiply two strings together
- Can't convert a "word" into a "number"
- Try it! The interpreter will let you know when you're wrong

For Loops

For loops iterate *through a collection* (like VB)

```
>>> for q in [2,'s', False]:
... type(q)
...
<class 'int'>
<class 'str'>
<class 'bool'>
```

For Loops

That makes them *very* versatile! But what if we just want to count to 3...

```
>>> for i in range(4): #creates a list of ints 0-3
...         print(i)

0
1
2
3

>>> for i in range(2,4): #define a start and end
        print(i)

2
3
```

For Else

You can add a final command to the end of a for loop, once the loop is done.

```
for i in range(3):
    print(i)
else:
    print("All done!")
```

Produces:

```
0
1
2
All Done!
```

While Loop

While loops are conditional, and they don't loop through a collection

```
>>> x=2
>>> while x < 4:
...         print(x)
...         x += 1


2
3
```

What about...

```
>>> while 2 < 3: #This loop will never end!
```

Generally common practice to use

```
>>> while True:
```

If, Elif, Else

Generally pretty much the same as other languages.

```
x=2
if x == 3:
    print("Three")
elif x == 4:
    print("Four")
else:
    print("Two") #this one happens!
```

Function Definitions

Functions, like variables, are not strictly typed. You do not need to declare a return variable, like in many other languages. Python trusts that you know when you want to return something or not.

```
def addTwo(a): #returns an int
    return a+2

def sayHello(): #returns nothing
    print("Hello!")

def sayGoodbye(name): #returns nothing
    print("Goodbye "+ name )
```

Function Example

Here's a little example of a "pythonic" function

```
def sumsq(*a): #pass any number of args
    return sum([x*x for x in a]) #sum of squares
```

Clean, simple and powerful - yet the logic can be followed. Uses lists, loops, operators, and built in functions.

```
sumsq(2,2,1)
9
sumsq()
0
sumsq(2)
4
```

Generators

Generators are like functions, but instead of *returning* a variable, they *yield* a variable.

```
def enumerate(seq): #actually built-in
 n = 0
 for item in seq:
 yield n, item
 n += 1
```

Builds an *iterator*, with a next method to keep going. These are pretty nifty if you can get them working.

Section 3

Classes

Where are We?

What Are Objects?

For those of you not familiar with Object Oriented Programming (OOP), here's a basic rundown:

- "Objects" are complex data structures that have some analog in the real world
- In Pyhton, *everything* is an object (even an integer), but you can create your own!
- Objects have many uses. Here's two:
  - Storing data with individual characteristics
  - Providing functions that interact with that object or are related to that object

A Class of its Own

Let's think about this again with an example

- Think of objects as a template for a physical thing. For example, *a Car*
- The object has things that describe it, *weight, color, cost, fuel level, etc*
- The object can **do** things, *drive, start, etc.*
- Each individual version of this object can have different values.

Our Example Class

Classes have a special function that is called whenever a copy of this class is created

```
class Car(): #Bracket for inheritance
    def __init__(self, weight):
        #every fn in a class has self
        self.weight = weight
        self.color = "FF0000" #hex color - why not?
        self.cost = self.weight*10
```

Create an *instance* of this car with

```
car = Car(1000)
car.color = "FFA500" #change color
```

Class Functions

You can have other functions that manipulate values of the instance

```
class Car(): #Bracket for inheritance
    def __init__(self, weight):
        ...
    def rgb_color(self):
        return tuple(ord(c) for c in self.color.decode(hex))
```